Runtime之KVO底层原理实现

1.KVO 是什么?
KVO 是 ios的一种设计模式：观察者模式，指定一个被观察者(比如Person类里面的name就是被观察者)，当被观察者的某个属性发生改变，会通知响应的回调，处理结果。
在 MVC 设计架构下的项目，KVO 机制很适合实现 mode 模型和 view 视图之间的通讯。
2.KVO原理

- 2.1 当一个属性被观察的时候，系统会在**第一次观察的时候创建一个派生类**，重写基类里面任何被观察属性的setter方法，派生类在被重写的setter里面实现**通知机制**
- 2.2 如果被观察的类为Perosn，那么派生类为NSKVONotifying_Perosn
- 2.3每个类都有一个isa指针指向本类，当一个类的属性第一次被观察的时候，那么系统会偷偷的把isa指针指向当前类的派生类，从而给被观察者setter的时候偷偷执行派生类的setter方法
- 2.4 键值观察通知依赖与NSObject的2个方法：WillChangeValueForKey 和 DidChangeValueFoKey；在一个被观察属性发生改变之前，willChangeValueForKey一定会被调用，这就 会记录旧的值；而当发生改变之后，didChangeValueForKey会被调用，继而 observeValueForKey:ofObject:change:context: 也会被调用。
-
- 补充：KVO的这套实现机制中苹果还偷偷重写了class方法，让我们误认为还是使用的当前类，从而达到隐藏生成的派生类

3.自己动手实现一个KVO思路
3.1 * 获取当前类，看有没有 keyPath 对应的setter 方法，没有，直接return
3.2 * 获取当前类名判断是不是 XXX_KV0 开头的，不是XXX开头的，就是第一次观察
3.3* 第一次观察，就得创建派生类，这拿个BOOL记录下，是不是第一次
3.4 * 检查派生类里面有没有keyPath 对应的setter 方法，没有，直接添加一个方法(这得写一个IMP方法地址，kvo_setter)
3.5 * BOOL为YES时候，注册这个派生类，并且把当前类的指针，指向派生类的指针
3.6 * 利用Runtime属性，给self添加2个属性，就是options和observer
3.7 * kvo_setter里面逻辑
* 保存当前KVO的类

* 将self的isa指针指向父类，调用父类setter方法
* 根据参数 _cmp 获取keyPath
* 调用父类setter方法，重新赋值
* 取出观察者
* 通知观察者，执行通知方法
* 重新修改为类

以下直接贴代码

```
//

#import "ViewController.h"
#import "Perosn.h"
#import "NSObject+HHKVO.h"

@interface ViewController ()

@property (nonatomic,strong) Perosn * person;

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    Perosn * person = [[Perosn alloc]init];
    [person HH_addObserver:self forKey:@"name" options:NSKeyValueObservingOptionNew context:nil];
    _person = person;
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary<NSKeyValueChangeKey,id> *)change
                       context:(void *)context
{

    if ([keyPath isEqualToString:@"name"]) {
        NSLog(@"name %@",self.person.name);
        NSLog(@"change %@",change);
    }
}
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [super touchesBegan:touches withEvent:event];
    self.person.name = [NSString stringWithFormat:@"%d",arc4random() % 100 + 1];

}
```

```
''

#import <Foundation/Foundation.h>

@interface NSObject (HHKVO)

- (void)HH_addObserver:(NSObject *)observer
                forKey:(NSString *)key
               options:(NSKeyValueObservingOptions)options
               context:(nullable void *)context;

@end

```

```

#import "NSObject+HHKVO.h"
#import <objc/message.h>


#define HHKVO_Name @"HHKVO"
#define HHKVO_Observer_Key "HHKVO_Observer_Key"
#define HHKVO_Changet_Key "HHKVO_Change_Key"

@implementation NSObject (HHKVO)

static NSString * getSetterMethodParameter(NSString * key){
    if (key.length == 0) {
        return nil;
    }
    NSString * newFirsRangetKeyName = [[key substringToIndex:1] uppercaseString];
    NSString * endRangeKeyName = [key substringFromIndex:1];
    NSString * methodString = [NSString stringWithFormat:@"set%@%@:",newFirsRangetKeyName,endRangeKeyName];
    return methodString;
}

static NSString * getterForSetter(NSString *setter)
{
    if (setter.length <=0 || ![setter hasPrefix:@"set"] || ![setter hasSuffix:@":"]) {
        return nil;
    }

    // remove 'set' at the begining and ':' at the end
    NSRange range = NSMakeRange(3, setter.length - 4);
    NSString *key = [setter substringWithRange:range];

    // lower case the first letter
    NSString *firstLetter = [[key substringToIndex:1] lowercaseString];
    key = [key stringByReplacingCharactersInRange:NSMakeRange(0, 1)
                                       withString:firstLetter];

    return key;
}


static void kvo_setter(id self,SEL _cmp,id newValue){

    NSString * keypath = getterForSetter(NSStringFromSelector(_cmp));

    // 保存当前KVO的类
    Class kvoClass = [self class];
    // 将self的isa指针指向父类，调用父类setter方法
    object_setClass(self, class_getSuperclass([self class]));

    // 调用父类setter方法，重新复制
    objc_msgSend(self, _cmp, newValue);
    // 取出观察者
    id objc = objc_getAssociatedObject(self, HHKVO_Observer_Key);

    NSNumber * optionsNumber = objc_getAssociatedObject(self, HHKVO_Changet_Key);
    NSDictionary * dict = @{@"new":newValue,@"kind":optionsNumber};
    // 通知观察者，执行通知方法
    objc_msgSend(objc, @selector(observeValueForKeyPath:ofObject:change:context:), keypath, self, dict, newValue);
    // 重新修改为KVO_Person类
    object_setClass(self, kvoClass);

}


- (void)HH_addObserver:(NSObject *)observer
                forKey:(NSString *)key
               options:(NSKeyValueObservingOptions)options
               context:(nullable void *)context
{
    /*1.创建一个派生类 */
    //1.1 获取类名
    NSString * className = NSStringFromClass([self class]);

    Class  newClass = object_getClass(self);

    BOOL hasClass = [className hasPrefix:HHKVO_Name];

    if (!hasClass) {
        //如果不包含 那么 就是第一次 监听

        //1.2 动态拼接类名
        NSString * newClassName = [HHKVO_Name stringByAppendingFormat:@"_%@",className];
        //1.3 动态创建类
        newClass = objc_allocateClassPair([self class], newClassName.UTF8String, 0);
    }

```

```objc
- (void)HH_addObserver:(NSObject *)observer
                forKey:(NSString *)key
               options:(NSKeyValueObservingOptions)options
               context:(nullable void *)context
{
    /*1.创建一个派生类 */
    //1.1 获取类名
    NSString * className = NSStringFromClass([self class]);

    Class  newClass = object_getClass(self);

    BOOL hasClass = [className hasPrefix:HHKVO_Name];

    if (!hasClass) {
        //如果不包含 那么 就是第一次 监听

        //1.2 动态拼接类名
        NSString * newClassName = [HHKVO_Name stringByAppendingFormat:@"_%@",className];
        //1.3 动态创建类
        newClass = objc_allocateClassPair([self class], newClassName.UTF8String, 0);
    }

    //1.3.1 获取 key 的setter 方法名字
    NSString * setMethodName = getSetterMethodParameter(key);
    //1.3.2 根据名字生成 方法
    SEL method = NSSelectorFromString(setMethodName);
     //1.3.3 检查类里面有没有 set方法
    Method setMethod = class_getInstanceMethod([self class], method);

    if (!setMethod) {
        // 如果当前类里面 没有 对应的方法 抛出异常
        NSString *reason = [NSString stringWithFormat:@"Object %@ does not have a setter for key %@", self, key];
        @throw [NSException exceptionWithName:NSInvalidArgumentException
                                       reason:reason
                                     userInfo:nil];
        return;
    }


    //1.3.4 如果没有的话，那就给类添加方法
    if (![self hasSelector:method]) {
         //1.3.5 给新的类赋值 set 方法，但是这里得先获取参数列表
        const char * types = method_getTypeEncoding(setMethod);
         //1.3.6 给新的类赋值 set 方法
        class_addMethod(newClass, method, (IMP)kvo_setter, types);
    }
    if (!hasClass) {
        //1.4 注册 这个 类
```

```objc

    //1.3.4 如果没有的话，那就给类添加方法
    if (![self hasSelector:method]) {
         //1.3.5 给新的类赋值 set 方法，但是这里得先获取参数列表
        const char * types = method_getTypeEncoding(setMethod);
         //1.3.6 给新的类赋值 set 方法
        class_addMethod(newClass, method, (IMP)kvo_setter, types);
    }
    if (!hasClass) {
        //1.4 注册 这个 类
        objc_registerClassPair(newClass);

        //2.0 更改当前 类的指针
        object_setClass(self, newClass);
    }
    //到这了 有一个问题，那就是 如何通知 系统的方法  那就是 给当前类 添加一个属性
    objc_setAssociatedObject(self, HHKVO_Observer_Key, observer, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    objc_setAssociatedObject(self, HHKVO_Changet_Key, @(options), OBJC_ASSOCIATION_ASSIGN);

}
- (BOOL)hasSelector:(SEL)selector{
    unsigned int count = 0;
    Method * methodList = class_copyMethodList([self class], &count);
    for (int i = 0; i < count; i++) {
        SEL sel = method_getName(methodList[i]);
        if (sel == selector) {
            break;
            return YES;
        }
    }
    return NO;
}

@end
```